# OS Lab 6

Om Patil (200010036)
Hrishikesh Pable (200010037)

February 2023

## 1 Transformations

### 1.1 Grayscale

This transformation converts a colour RGB image to a grayscale image. This simple transform can be achieved by taking the weighted average of the R, G and B channels. The exact formula for the same is as follows,

$$gray = red * 0.114 + green * 0.299 + blue * 0.587$$

### 1.2 Blur

This transformation blurs the image. This is done by running a 5x5 convolution filter across the image five times. Running a convolution involves taking a specific weighted average of the pixels around the pixel to be calculated. The filter or kernel determines the value of these weights. The exact kernel values used by us to blur are as follows,

$$\frac{1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

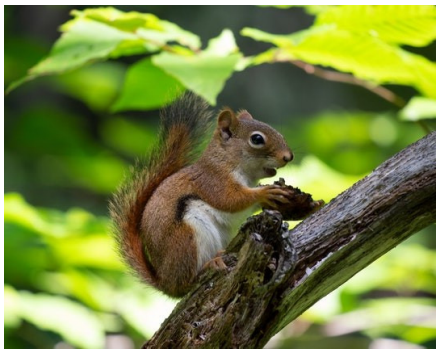
## 2 Correctness

The way that we have implemented each of the tasks in part 2, it is expected that the data is sent in order. However, to verify this, we devised a method. We compare the output image using the diff command.

```
diff -s part1-output.ppm part2-output.ppm
```

If the transfer was done correctly, i.e. in order, then there is no difference between the output from the part 1 code and the output from the part 3 code. Using this method, we verified that our approach transferred pixels in order.

# 3 Sample Outputs



(a) *Original*

(b) *Transformed*

Figure 1: Sample 1

(a) *Original*

(b) *Transformed*

Figure 2: Sample 2

Figure 1, 2 & 3 show a few sample inputs and their corresponding transformed outputs. As it is clearly visible, the image is converted to grayscale and blurred.

# 4 Analysis

The run-times for the various methods are as follows,

The data shows that the run times of the multi-threaded/process methods take longer. This is because the synchronisation overhead of the multi-threading/process methods is more significant than its advantage.

(a) *Original*

(b) *Transformed*

Figure 3: Sample 3

| Image Size | Part 1 | Part 2.1a | Part 2.1b | Part 2.2 | Part 2.3 |
|---|---|---|---|---|---|
| 535x426 | 0.098421 | 1.061387 | 0.153337 | 0.199185 | 0.197207 |
| 640x480 | 0.138908 | 0.194791 | 0.213934 | 0.261313 | 0.258504 |
| 493x356 | 0.082997 | 0.994429 | 0.115394 | 0.148183 | 0.143738 |

Table 1: Runtimes in seconds

# 5 Implementation

## 5.1 Threads synchronised using atomic operations

This implementation was quite easy. It involved implementing a simple spin-lock to ensure the atomicity of the critical section. As the code was simple not much debugging was required.

## 5.2 Threads synchronised using semaphores

This implementation was also quite easy. It involved implementing semaphores to ensure the atomicity of the critical section. Compared to the last method, the code was slightly more involved and required slightly more debugging.

## 5.3 Process via shared memory

This implementation was the hardest as we had to transfer the processed pixels through shared memory to the other process. This require us to synchronize the transfer and ensure each pixel is read correctly. We had to debug this the most.

## 5.4 Process via pipes

This implementation was quite involved but not as much as the last one. We had to pass the data from the first process running the first transformation to

the other process keeping track of what data was received and synchronizing the processing of the two processes. We needed to do some debugging to get the synchronization to work properly.